Java 5 Concurrency

Part One

SE 441

Prof. Bullinger
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Threads

• Pros
  • Easy to understand
  • Easy to implement

• Cons
  • Cannot reuse a thread
  • Management overhead
  • No inherent resource management
  • Potential stability issues under load
Synchronization

- **Pros**
  - Easy to understand
  - Easy to implement

- **Cons**
  - High overhead
  - All-or-nothing semantics - no timeouts
  - No metrics or profiling of locking scenarios
  - Course-grained locking
Java 5 Concurrency

• Problems with Java threads
• Synchronized Collections
• Concurrent Collections
• Synchronizers
• Tasks
• Executor Framework
• Cancellation and Shutdown
Synchronized vs. Concurrency

- Synchronized ensures thread safety
  - May degrade to sequential operation
  - Counter to concurrency needs

- Concurrency
  - Parallel execution of “tasks”
  - Optimize liveness
  - Limited by thread safety?
Synchronized Collections

- e.g. Vector, Hashtable, List
- Encapsulates state
- Provides synchronized methods for public access
- Thread safe?
  - Data integrity is maintained
  - Behavior may not be consistent
Synchronized Collections

• Requires client-side locking for compound actions
  • Iteration
  • Ordered navigation
  • Put-if-absent
  • Check-then-act

• Client-side locking may not be sufficient
Interleaved Actions

- Different threads access the same collection:

```java
// retrieve the last member
public static Object getLast(Vector memberList) {
    synchronized (memberList) {
        int lastIndex = memberList.size() - 1;
        return memberList.get(lastIndex);
    }
}
```

```java
// remove the last member
public static Object deleteLast(Vector memberList) {
    synchronized (memberList) {
        int lastIndex = memberList.size() - 1;
        return memberList.remove(lastIndex);
    }
}
```

```java
// process each member of the committee
for (int memberCount = 0; memberCount < memberList.size(); memberCount++) {
    doSomething(memberList.get(memberCount));
}
```

- throws ArrayIndexOutOfBoundsException exception!
Interleaved Actions

• Iterate with an iterator:

```java
// process each member of the committee
for(Member member : memberList)
{
    doSomething(member);
}
```

• throws ConcurrentModificationException!
Hidden Iterators

- Implicit iterators invoked through other methods
  - `toString()` on collections
  - Collections passed as arguments
  - Collection methods
    - `hashCode`
    - `equals`
    - `containsAll`
    - `removeAll`
    - `retainAll`
Implications

- Thread-safety
  - Guaranteed by the synchronized collections
- Reasonable behavior?
  - Requires extensive client-side locking
  - Serializes access to content
  - Requires understanding of potential interleave issues
  - Contrary to liveness and concurrency
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Concurrent Collections

• Provides thread-safety in addition to reasonable concurrent behavior
  • Offers dramatic performance improvements with no risk

• Replacement classes:
  • ConcurrentHashMap
  • CopyOnWriteArrayList

• New classes:
  • Queue
  • Deque
Concurrent Collections

- Methods include new atomic operations:
  - putIfAbsent
  - removeIfEqual
  - replaceIfEqual
  - etc.
- Does not support client-side locking
- Cannot lock the entire collection!
Concurrent Collections

• Weakly-consistent iterators
  • Traverses the state of the collection when the iterator was constructed
  • Tolerates concurrent modifications
  • May reflect modifications to the collection
  • `size()` and `isEmpty()` methods are approximations!

• Optimized for access methods:
  • `get`
  • `put`
  • `containsKey`
  • `remove`
ConcurrentHashMap

•Uses lock striping for locking

•Divides the hash buckets into 16 groups

•Locks each group of buckets independently

•Supports an arbitrary number of readers

•Supports 16 concurrent write operations

•Can be increased if proven necessary

•Iterators do not throw ConcurrentModificationException

•Weakly-consistent iterators
CopyOnWriteArrayList

- Replacement for synchronized collections:
  - List -> CopyOnWriteArrayList
  - Set -> CopyOnWriteArraySet
- No need to lock or copy during iteration
- Implemented as effectively immutable collection
  - Modifications transform/copy rather than mutate
- Iterators reflect the state of the collection when the iterator is created
- Expensive for large collections
  - Use when iteration more common than modification
BlockingQueue

- Standard queue behavior
- Standard Collection behavior (?)
- Can be bounded or unbounded
- Method summary:

<table>
<thead>
<tr>
<th></th>
<th>Throws exception</th>
<th>Special value</th>
<th>Blocks</th>
<th>Times out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>add(e)</td>
<td>offer(e)</td>
<td>put(e)</td>
<td>offer(e, time, unit)</td>
</tr>
<tr>
<td>Remove</td>
<td>remove()</td>
<td>poll()</td>
<td>take()</td>
<td>poll(time, unit)</td>
</tr>
<tr>
<td>Examine</td>
<td>element()</td>
<td>peek()</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
BlockingQueue

- Ideal for producer-consumer patterns:
  - `put()` and `take()` methods block, throttling thread behaviors
  - Works with multiple producers and multiple consumers
- Implements safe publication of objects
- Transfer ownership for serial thread confinement
BlockingQueue

- Multiple implementations provided:
  - `LinkedBlockingQueue`, `ArrayBlockingQueue`
    - FIFO queue
  - `PriorityBlockingQueue`
    - queue with (arbitrary) prioritization
  - `SynchronousQueue`
    - Queues the threads, not the data!
    - Effective only when producer rate \( \leq \) consumer rate
Deque

• A double-ended queue
• Supports insertion and removal at both ends
• Method summary:
Deque

• When used as a queue:

<table>
<thead>
<tr>
<th>Queue Method</th>
<th>Equivalent Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(e)</td>
<td>addLast(e)</td>
</tr>
<tr>
<td>offer(e)</td>
<td>offerLast(e)</td>
</tr>
<tr>
<td>remove()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>poll()</td>
<td>pollFirst()</td>
</tr>
<tr>
<td>element()</td>
<td>getFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>

• When used as a stack:

<table>
<thead>
<tr>
<th>Stack Method</th>
<th>Equivalent Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(e)</td>
<td>addFirst(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>
Work-stealing

• Given multiple work deques for multiple consumers:
  • When a given consumer’s work queue is empty, it can remove an element of work from the tail of another’s deque
  • Under normal load, contention is reduced as each consumer has its own deque
  • When stealing, the consumer extracts from the tail instead of the head of the deque, reducing contention
• Well-suited when consumers are also producers
  • Graph-traversal
  • Consumers add work back into the deque
Blocking and Interruption

• Reasons for blocking
  • I/O
  • Lock acquisition
  • Sleep
  • wait / notify

• Some blocking methods throw InterruptedException
  • Indicates premature conclusion of a blocking method
  • Implies attempted cleanup when interrupted
Blocking and Interruption

• Interruption is a cooperative mechanism
  • Interrupts are only a request, not a demand
  • Client code must trap, then gracefully handle interruptions
  • Interruption semantics should be limited to cancellation only

• Two responses to interruption:
  • Propagate the InterruptedException
  • Restore the interrupt
Blocking and Interruption

- Propagate the InterruptedException:
  - Don’t catch the exception
    - Force the client to catch it
  - Catch and re-throw
    - Allows context-specific cleanup
- Restore the interrupt
  - Cannot throw an exception as a Runnable
  - Explicitly restore the interrupt status
Restoring an Interrupt

```java
public class TaskRunnable implements Runnable {
    BlockingQueue<Task> queue;

    public void run() {
        try {
            processTask(queue.take());
        } catch (InterruptedException exception) {
            // restore the interrupt status
            Thread.currentThread().interrupt();
        }
    }
}
```
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Synchronizers

- Coordinate flow of control between producers and consumers
  - Based on the state of the synchronizer
  - Determines whether threads arriving at the sync point should wait or continue
  - Allows manipulation of the gating state
  - Provides efficient wait mechanism
Blocking Queues

• Synchronize threads based on availability of data or space in the queue / deque

• `take()`

• `put()`
Latches

• Threads cannot pass until the terminal state is achieved
  • Wait until all resources are available
  • Ensure all services are available
  • Wait for all participants
• A one-shot semantics
CountDownLatch

• Initialized with a given count
• The countDown() method decrements the count
• The await() method blocks until the count is zero
• All participating threads call countDown(), then await()
  • The await method has an optional timeout
• Can be binary
• Resettable latches are barriers
Exercise

- Modify ThreadMadness to use a CountDownLatch to detect the end of the simulation
FutureTask

• A latch that gates a thread based on a result of an asynchronous operation

• Results are available only when complete
  • The get( ) method blocks!

• Three possible states
  • Waiting, running, complete

• Requires a Callable client (more on this later)

• Can be cancelled

• Safe publication of results is guaranteed
Semaphores

• Maintains a finite set of *permits* for an operation or resource

• The acquire( ) method requests a permit and blocks
  • Acquire has an optional timeout parameter

• The release( ) method adds a permit

• A semaphore with one permit can be used as a mutex
  • The permit can be released by another thread!

• Can be initialized as *fair* to ensure FIFO semantics on requests
Barriers

• Waits for threads to join at the same time
  • The barrier is created with a defined number of parties / participants
  • Threads call `await()` on the barrier
    • `Await` has an optional timeout
  • `await()` blocks until all parties have joined
  • Provides for an optional Runnable called when the barrier is released
  • `Await` returns an arrival index to the caller

• `CyclicBarrier` is a reusable barrier
Exercise

• Update ThreadMadness to use a CyclicBarrier to detect the end of the simulation
Exchangers

• A synchronization point where two threads can exchange objects

• Threads call exchange( ) with an object

• Exchange has an optional timeout

• Each thread blocks waiting for the other thread

• The object is returned to the other thread when both threads arrive

• Ensures safe publication of objects between threads
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Tasks

• Definition: An independent activity

• Abstract / discrete units of work

• Minimal or no dependencies on other tasks

• Can be executed in parallel

• Represents a fraction of processing capability of the platform

• Optimizes throughput and responsiveness

• Exhibits graceful degradation
Task Execution Policies

• Separation of task submission and execution
• Define what, where, when and how a task is executed
  • Sequential execution
  • Thread-based execution
  • Executor Framework
Sequential Task Execution

```java
// Single threaded web server
class SingleThreadWebServer {
    public static void main(String[] args)
        throws IOException {
        ServerSocket socket = new ServerSocket(80); // http server
        while(true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```
Consequences?

- Limited to processing a single connection at a time
- New connections must wait
- I/O-bound performance
- Cannot exploit multiple CPUs
- Not practical
Threaded Task Execution

```java
// threaded task web server
class ThreadPerTaskWebServer {
    public static void main(String[] args)
        throws IOException
    {
        ServerSocket socket = new ServerSocket(80); // http server
        while(true)
        {
            final Socket connection = socket.accept();
            Runnable task = new Runnable()
            {
                public void run()
                {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```
Consequences?

• Task processing is offloaded from the main thread
• Incoming connections are handled immediately
• Tasks are processed in parallel
  • Much better performance when I/O bound
• handleRequest must be thread-safe
• Limitations?
  • Thread lifecycle overhead
  • Resource consumption / limitation
  • Stability under heavy load
Task Execution

• Another mechanisms is required that:
  • provides parallelism of task execution,
  • scales to the dynamic limits of the platform, and
  • degrades gracefully under load.
Result-bearing Tasks

• Runnable does not support a return value
• Runnable does not throw any exceptions
• This complicates task synchronization
• Many tasks are deferred computations with a specific result
• The Callable interface provides a Runnable with a return value and support for exceptions
Future

- An abstraction for the lifecycle of a task
- Provides client access to a task’s state and results
  - cancel - cancels execution of the task
    - provides an interrupt option parameter
  - get - retrieves the results of the task
    - with optional timeout to limit task execution time
- Returned from the submit method of an Executor
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Executor Framework

• The basis of task execution in Java 5+
• Maps tasks onto threads
• Defines the task execution policy
• Decouples task submission from task execution
• Ideal for all producer-consumer patterns

```java
public interface Executor {
    void execute(Runnable command);
}
```
Executor Example

```java
// Executor-based task web server
class TaskExecutionWebServer {
    private static final int MAX_THREADS = 100;
    private static final Executor taskExecutor = Executors.newFixedThreadPool(MAX_THREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80); // http server
        while(true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            taskExecutor.execute(task);
        }
    }
}
```
Example Executors

```java
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable runnable)
    {
        runnable.run();
    }
}
```

```java
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable runnable)
    {
        new Thread(runnable).start();
    }
}
```
Execution Policies

• Define the parameters around task execution:
  • What thread will be used?
  • In what order will the tasks execute?
  • How many tasks can run concurrently?
  • How many tasks can be queued for running?
  • How are tasks rejected?
    • Which tasks?
    • How is the client notified (if at all?)
  • Are there any task pre or post-execution behaviors?
Task Lifecycle

• Tasks with an Executor have four states
  • Created, submitted, started, and completed
  • Submitted tasks can be cancelled
  • Started tasks can be interrupted
Thread Pools

• A bounded work queue holding tasks waiting to execute

• A limited number of worker threads execute the tasks
  • Request a task from the work queue
  • Execute the tasks
  • Wait for another task
Thread Pools

• Enables reuse of threads
  • Minimize overhead
• Reduced task latency (when the queue is empty)
• Ability to dynamically tune the thread pool behavior
Thread Pools

• FixedThreadPool
  • Creates threads as need for tasks up to a fixed limit
  • Maintains a constant maximum number of threads

• CachedThreadPool
  • Dynamically alters the number of worker threads based on the number of tasks in the queue
  • No bounds on the number of threads
Thread Pools

- SingleThreadExecutor
  - One thread processing tasks in sequential order
  - Order is guaranteed

- ScheduledThreadPool
  - A fixed number of threads supporting delayed and periodic task execution
  - A replacement for Timer!
    - Timer uses a single thread for all clients
Moving Forward

• The executor framework should be used for all multi-threaded applications

• Explicit thread creation and start should be replaced!
Exercise

• Update ThreadMadness to utilize a default Executor to run the simulation threads
Executor Lifecycle

- Executors are easy to start
- Executors support submission of tasks
  - Runnable
  - Callable
- Task submission returns a *Future*
- Tasks submission and Future results constitute safe publication
Executor Lifecycle

• Executors must be shutdown too!
  • Tasks may be in the queue, in process, or completed
  • Two philosophies for shutdown
    • Graceful - complete current processes and queue
    • Rude - interrupt current processes
  • Three states of an Executor:
    • running, shutting down, terminated
Executor Lifecycle

• ExecutorService methods:

  • shutdown - no new tasks are accepted, waiting and running tasks are completed

  • shutdownNow - no new tasks are accepted, running tasks are cancelled, waiting tasks are not started

  • awaitTermination - the caller blocks until the service is complete
Java 5 Concurrency

- Problems with Java threads
- Synchronized Collections
- Concurrent Collections
- Synchronizers
- Tasks
- Executor Framework
- Cancellation and Shutdown
Cancellation and Interruption

- Tasks are typically run-to-completion semantics
- Stopping a thread externally is not safe
- Cancellation of a task must be cooperative!
  - Complete any work in progress
  - Ensure the consistency of any state
  - Release resources
  - Notify any interested clients
- Failure, shutdown and cancellation should be considered up-front
Task Cancellation

• “An activity is cancelable if an external stimulus can move it to completion before its normal completion.” (JCiP)

• User-requested cancellation

• Time-limited activities

• Application events

• Errors / Failures

• Shutdown
Cancellation Policies

• Specify the how, when and what of cancellation behavior

• How is the task cancelled?

• When is cancellation permitted? (cancellation points)

• What is the task response to cancellation?

• Each task should adhere to a defined policy for cancellation

• Create a pattern for use throughout your application
Task Cancellation

• There is no way to preemptively stop a thread in Java

• Multiple mechanisms are available:
  • Cancellation request flag
  • Interruption
  • Poison Pill
Cancellation Request Flag

• A task defines a volatile boolean data member for a request flag

• The task provides public access to set the request flag value

• The task loop examines the state of the request flag periodically

• The task loop exits when the request flag is set

• What about blocking calls within the task?
Interruption

- Each Java thread has an interrupted status
- Interrupting a Thread does not necessarily affect its behavior
- An interrupt is only a request to the Thread
- Methods are provided to test and reset the interrupted status

<table>
<thead>
<tr>
<th>void</th>
<th>interrupt()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interrupts this thread.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static boolean</th>
<th>interrupted()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests whether the current thread has been interrupted.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>boolean</th>
<th>isInterrupted()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests whether this thread has been interrupted.</td>
<td></td>
</tr>
</tbody>
</table>
Interruption

- A thread interrupt is merely a cancellation flag implemented by the thread!

- Many Java libraries with blocking calls support interrupts via threads (i.e. `Thread.sleep()`)

- Non-blocking threads must query and react to interrupts (similar to the libraries)

- “Interruption is the most sensible way to implement cancellation.” (JCiP)
Interrupt Policies

- Threads should have an interrupt policy:
  - How does the thread interpret the interrupt request?
  - What units of work are considered atomic?
    - Which defines the cancellation points
  - What is the desired / required timeliness of the interruption?
- For thread pools, the task implements the interrupt policy
Detecting Interrupts

• Two options for detecting interrupts:
  • Try-catch around blocking calls
  • Periodic examination of thread interrupt status

• Two options for reacting to interrupts:
  • Propagate (or throw) an exception to the caller
  • Handle the interrupt internally and restore the interrupt status
    • Avoid infinite loops caused by resetting the status inside a try-catch block!
Interruption Caveats

• Know a task / thread interrupt policy before interrupting it!

• Required good documentation for application code

• General-purpose application and library code should never swallow interrupts

• This should only happen in interrupt policy code
Imposing Interrupts

• Tasks can be interrupted via ExecutorService
  • With an optional flag to interrupt if running
• Tasks can be cancelled through their Future representation
  • Don’t interrupt a thread in a thread pool, the affected task is unknown!
Non-interruptible Blocking

• Some library methods do not support interrupts when blocked
  • Socket I/O - close the socket when cancelled
  • Synchronous I/O - close the channel when cancelled
  • Asynchronous I/O - close or wakeup the channel
• Lock acquisition - use explicit locks!
Poison Pills

• A sentinel value / object placed in a queue
  • Notifies the producer to terminate
  • Signals the consumer to terminate
    • With FIFO, ensures pending work completes

• Ideal for simple systems
  • Does not scale
  • Works on unbounded queues only (no blocking!)
Canceling Thread Services

- Only the owner of a thread should cancel it
- ExecutorService provides shutdown methods
  - Provide cancellation policies in your own implementations of Executor services
Abnormal Thread Termination

- RuntimeExceptions can destroy a running application
- Consider a small framework to trap uncaught exceptions gracefully

```java
// exception-safe framework
public void run() {
    Throwable thrown = null;
    try {
        while(!isInterrupted())
            runTask(getTaskFromWorkQueue());
    } catch(Throwable exception) {
        thrown = exception;
    }
    finally {
        threadExited(this, thrown);
    }
}
```

- Use the UncaughtExceptionHandler interface

```java
public interface UncaughtExceptionHandler {
    void uncaughtException(thread thread, Throwable exception);
}
```

- Use both techniques for long-running applications
Shutdown Hooks

- Register a task with the Runtime class
  ```java
  void addShutdownHook(Thread hook)
  Registers a new virtual-machine shutdown hook.
  ```

- On JVM shutdown, all registered shutdown hooks are started

- The order of execution is non-deterministic
  - Don’t embed timing dependencies across multiple hooks!

- Ideal for service or application cleanup
Example Shutdown Hook

```java
public void start() {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            try {
                LogService.this.stop();
            } catch(InterruptedException ignored) {} 
        }
    });
}
```
Daemon Threads

• The JVM supports two types of threads:
  • Normal
  • Daemon

• When a thread exits, the JVM inventories all remaining threads
  • If no normal threads remain, the JVM exits
  • Daemon threads do not prevent the JVM from exiting

• Use Daemon threads for background tasks, and only sparingly!
Finalize

• Finalizers run in a JVM-managed thread
  • i.e. garbage collection
• Finalizers must be thread-safe
• Finalizers have no guarantee of execution
• Avoid using `finalize` methods
  • Use `finally` and explicit `close` instead